http://www.wkap.nl/journalhome.htm/0928-8910
This article appears in Vol 8, No 1, Jan 2001, pp 31-61.

Automatic Generation of Test Oracles - From Pilot Studies to Application

Martin S. Feather
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 354 1194
Martin.S.Feather@Jpl.Nasa.Gov

Ben Smith
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 353 5371
Ben.D.Smith@Jpl.Nasa.Gov

Abstract

We describe a progression from pilot studies to development and use of domain-specific verification and validation (V&V) automation. Our domain is the testing of an AI planning system that forms a key component of an autonomous spacecraft. We used pilot studies to ascertain opportunities for, and suitability of, automating various analyses whose results would contribute to V&V in our domain. These studies culminated in development of an automatic generator of automated test oracles. This was then applied and extended in the course of testing the spacecraft's AI planning system.

(Richardson et al, 1992) presents motivation for automatic test oracles, and considered the issues and approaches particular to test oracles derived from specifications. Our work, carried through from conception to application, confirms many of their insights. Generalizing from our specific domain, we present some additional insights and recommendations concerning the use of test oracles for V&V of knowledge-based systems.

Keywords: Testing, Test Oracles, Verification and Validation, Analysis, Planning, Autonomous Systems, NASA

1. Introduction

Cost, performance and functionality concerns are driving a trend towards use of self-sufficient autonomous systems in place of human-controlled mechanisms. Verification and validation (V&V) of such systems is particularly crucial given that they will operate for long periods with little or no human supervision. Furthermore, V&V must itself be done at low cost, rapidly and effectively, even as the systems to which it is applied grow in complexity and sophistication.

Spacecraft – especially deep space probes – exemplify these concerns. We have been involved in V&V of an AI planner that is a key component of a spacecraft's autonomous control system. In (Feather & Smith, 1998) we reported our use of an automated generator of automated test oracles to support these V&V activities. The paper is organized to show the progression of steps we followed leading up to this application, and the lessons we have learnt by reflecting upon our experience:

- First pilot study: rapid automated analysis (Section 2). In this study we determined the viability of a rapid analysis approach. We did case studies of two kinds of traditional design information, yielding confirmation of the viability of the analysis method for this kind of information.
- Target application – V&V of an autonomous spacecraft's planner (Section 3). The spacecraft autonomy is described, the needs for V&V of its planner component are described, and the overall approach to its V&V is outlined.
- Second pilot study: feasibility study of rapid analysis approach to V&V of the spacecraft planner (Section 4). We needed this second study to determine suitability of the rapid analysis approach to, specifically, checking plans generated by an AI planner. Particular concerns were scalability of the approach, and investment of domain experts' time. This pilot study produced instances of automatic test oracles in order to alleviate these concerns.
- Development of automated generator of planner test oracles (Section 5). Based on the lessons learned from the second

pilot study, we committed to developing a tool to be used in actual spacecraft testing. The tool would go beyond the capabilities of the second pilot study by both extending aspects of the analyses performed, and automating the generation of the test oracles themselves.

- Extension of the oracle to also validate requirements (Section 6).
- Application to V&V of spacecraft planner (Section 7). Metrics from the oracle's use in planner testing, and a discussion of the positive impact the oracle had on the testing effort.
- Lessons learned (Section 8). We describe lessons learned for both software engineering and V&V:
  - *Our experience re-iterates several well-understood virtues of pilot studies as a precursor to actual development.*
  - *When domain experts' time is a critical resource, follow an "on-demand" policy of knowledge acquisition.*
  - *V&V can make good use of redundancy and rationale, to increase assurance in the V&V results, and to assist in the development of the V&V technology itself.*
  - *The use of a database as the underlying analysis engine has practical applications and benefits.*
  - *Test oracles should yield results with far more content and structure than simply "passed" or "failed".*
  - *Translation between notations is a recurring need, and ideally should be done in such a way as to support understanding, specification and maintenance by domain experts.*
- Conclusions (Section 9). We summarize the relationship of our work to other efforts, and point to areas we believe are worthy of additional attention.

2. First pilot study: rapid automated analysis

The first stage was a pilot study that investigated analysis of simple properties of spacecraft designs. This was conducted in early 1997, primarily by the first author who, while not an expert in spacecraft, had access to spacecraft design documents and spacecraft experts. The purpose of this first study was to answer the following question:

Could simple analyses of spacecraft design information be performed rapidly by using a database as the underlying reasoning engine?

The approach under investigation was founded upon the use of a *database* as the underlying reasoning engine. We used AP5 (Cohen, 1989), a research-quality advanced database tool developed at the University of Southern California. AP5 combines ideas from AI, databases and programming languages. It has the flavor of Prolog, in the sense that it maintains a database of logical facts, in terms of which more complex logical definitions can be constructed. Logical queries against this fact base are automatically optimized in a database-like query-optimization manner. Finally, AP5 sits on top of CommonLisp, so has access to the full power of a programming language environment.

To use AP5 for design analysis, design information was stored as facts in the database, and analysis questions were coded as queries against that database of facts. The architecture of this approach is shown in Figure 1. Its four main steps were:

1. Manual creation of a database schema to represent the design information.
2. Loading the design information into the database. This was made a predominantly automated operation, by constructing special-purpose programs to extract information from design documents and translate into the format of the database schema. Automation made the approach practical for handling voluminous amounts of design information.
3. Determining V&V conditions and expressing them as database queries.
4. Analysis, performed by evaluating the V&V conditions as database queries against the data. The reporting of the query results was organized into confirmations and anomaly reports.

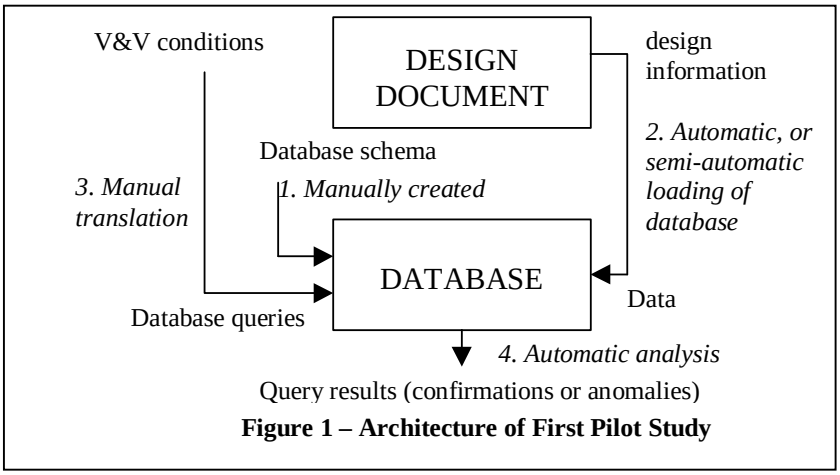


Figure 1 – Architecture of First Pilot Study

The pilot study examined two sets of design documents – interface diagrams (i.e., summaries of incoming and outgoing connections of software modules) and test logs (i.e., traces of behaviors generated in testing of the software components in simulations). Modest verification conditions were rapidly and successfully analyzed in this manner.

2.1. Example from first pilot study

This section presents some details taken from the first pilot study, illustrating the database-based approach to rapid automated analysis. The study dealt with a set of software module interface diagrams that portray message flow among the software modules of a spacecraft control system. The V&V conditions to be checked were various consistency conditions over this set of diagrams.

Figure 2 shows the top portion of an interface diagram of the kind analyzed in this pilot study. The rectangle in the center represents the **Smart Executive** software module. The solid incoming arrows on the left represent message types that can be received by that module (e.g., **MI_MODE_UPDATES**). These are annotated with the parameters (if any) of that message inside angled brackets (e.g., **updates**) and the name of the module from which they originate inside parentheses (e.g., **MIR**). The solid outgoing arrows on the right represent message types that can be produced by that module, with the module(s) to which they go listed in parentheses (e.g., **ACS**, **MIR**). The dotted arrows crossing the interior of the module represent cause-and-effect. A left-to-right cause-and-effect arrow indicates the receipt of the incoming message may lead to the production of the output message by the module (e.g., receipt of the **INITIALIZE** message by the **Smart Executive** module may lead to its production of a **MI_SEND_ALL_MODES** message). A right-to-left cause-and-effect arrow indicates the sending of the outgoing message may, via the actions of other modules, lead to the receipt of the incoming message (e.g., sending of a **MI_SEND_ALL_MODES** message by the **Smart Executive** may lead to receipt of a **MI_MODE_UPDATES** message from the **MIR** module).

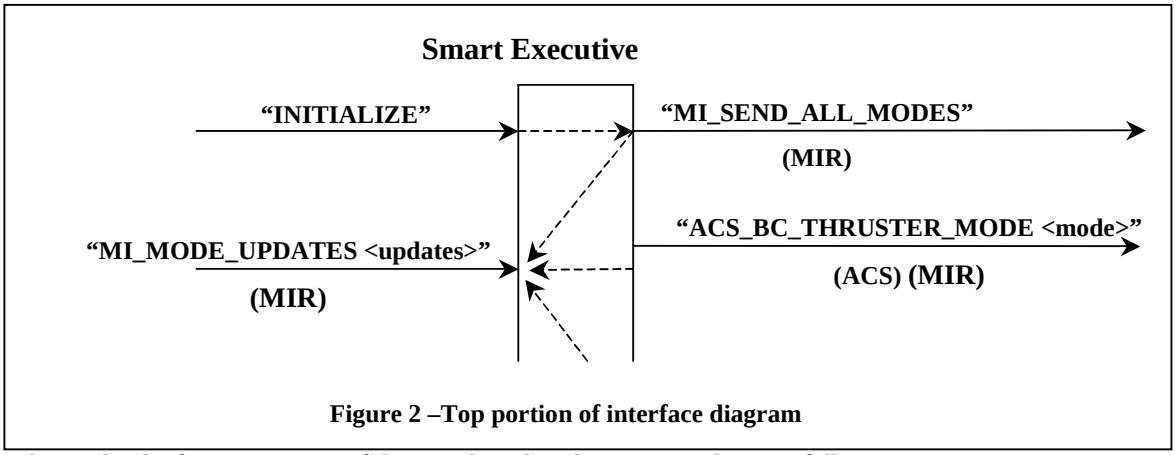


Figure 2 –Top portion of interface diagram

For this pilot study, the four main steps of the AP5-based analysis approach are as follows:

1. **Creating the database schema.** In this step, a straightforward database schema is manually created. The schema defines the AP5 representation to hold the information contained within interface diagrams. While the details are AP5-specific, the general ideas would apply regardless of the database used.

AP5 deals with objects and relations among objects. To represent the modules whose interfaces are drawn in these diagrams, an object type is defined. To represent the arrows within the interface diagrams, another object type is defined. To represent the message type conveyed by an arrow, a binary relation is defined between objects of type arrow and objects of type string (the name of the message type). To represent the destination module(s) to which an arrow points another binary relation is defined to relate an object of type arrow to an object of type string (the name of the destination module). Note that the aforementioned relations use string as the second type. This allows the representation of flawed diagrams (for example, an arrow naming a non-existent module as its destination).

Examples of AP5 definitions:
`(defrelation module :arity 1)` defines `module` to be a unary relation, used to represent module objects.
`(defrelation arrow-to-module-name :types (arrow string))` defines `arrow-to-module-name` to be a binary relation, used to relate an arrow to the name of a module to which it points.

2. **Loading the design information.** The interface diagrams' information content has to be entered as data into the AP5 database. In the pilot study, these design documents were available as postscript files, so a partially automated approach was followed to convert them into AP5 data. Custom Emacs macros were defined and manually applied to manipulate the postscript files, ultimately yielding code to make the appropriate insertions into the AP5 database.

   For example, the presence of the arrow labeled **MI_SEND_ALL_MODES (MIR)** was manipulated into a call to a Lisp function which would insert a new arrow object into the database, insert the `arrow-to-module-name` relation to hold between that arrow object and the string "**MIR"**, etc.

   This hybrid approach of some manual activities and some custom automation (e.g., Emacs macros and Lisp functions) was intended to achieve the data entry as rapidly as possible. Since development of automation itself takes time, it is necessary to gauge whether it is worthwhile as compared to simply entering information manually. In this pilot study there were 15 or so diagrams, with a typical diagram holding 30 or so arrows, so some modest automation was warranted. Judicious development of customized automation to supplant manual activities is a recurring theme of this entire approach.

3. **Expressing V&V conditions as queries.** An example of a straightforward V&V condition for a set of interface diagrams is: *for every arrow's destination module name(s) the corresponding interface diagram exists*.

   In AP5, this is condition is expressed as the following predicate:

```
(A (arrow1 str1)
    (implies
        (arrow-to-module-name arrow1 str1)
        (E (module1)
             (module-name module1 str1))))
```

   The syntax `(A (x) …x…)` indicates universal quantification, with `x` as the existentially universally quantified variable. Existential quantification is similarly expressed: `(E (x) …x…)`

   AP5's database-like nature comes into play in the evaluation of such queries. AP5 itself determines in which order to search the database contents so as to answer the query. The user is free to concentrate on the logical expression of the condition, leaving it to AP5's query optimization to handle efficient evaluation.

4. **Analysis and reporting.** Analysis was achieved by evaluating the AP5 queries encoding the V&V conditions against the AP5 database. Simple Lisp code suffices to print the results from these analyses. Typically one wants to know not only whether a V&V condition holds, but also the pertinent details of any violations of that condition. For example, if a diagram contains an arrow that names a non-existent module as its destination, it would be useful to print the module in which that arrow occurs, the message type conveyed by that arrow, and the name of the (non-existent) module to which it points.

2.2. Applications of the first pilot study

This first pilot study applied the analysis approach to two very different sets of design information:

- **Interface diagrams.** There were 15 or so interface diagrams, where a typical diagram held 30 arrows. The following consistency checks were evaluated across this set of diagrams:
  1. Every incoming arrow annotated as having come from an interface diagram must match in name and parameter types with a corresponding outgoing arrow on that diagram.
  2. Every outgoing arrow annotated as going to an interface diagram must match in name and parameter types with a corresponding incoming arrow on that diagram.

3. For every right-to-left cause-and-effect arrow, there must be a chain of outgoing and incoming message arrows, and left-to-right cause-and-effect arrows, that leads from the destination of that arrow back to the source of that arrow (e.g., for the right-to-left cause-and-effect arrow that links MI_SEND_ALL_MODES to MI_MODE_UPDATES, such a chain would exist if on the MIR interface diagram, a left-to-right cause-and-effect arrow led from the incoming MI_SEND_ALL_MODES arrow to its outgoing MI_MODE_UPDATES arrow).
4. For every chain of outgoing and incoming message arrows, and left-to-right cause-and-effect arrows, that start with an outgoing arrow on an interface diagram and end with an incoming arrow on that same diagram, there must be a left-to-right cause-and-effect arrow linking the incoming and outgoing arrows on that diagram.

Checks 1 and 2 were standard, simple checks of type correctness across the set of interface diagrams. Checks 3 and 4 were problem-specific checks of a notion of "causality". Note that these checks involve tracing chains of arrows. AP5's capability to define the transitive closure of a binary relation was ideally suited to expressing these chains.

- **Logs of message passing generated during test runs** (i.e., traces of behaviors generated in testing of the software components in simulations). The status of the spacecraft could be inferred from these logs, permitting safety conditions on the state of the spacecraft to be checked. For example, the attitude control was required to be in a special mode whenever the engine was not thrusting, so a message log that contained a message to turn off engine thrusting had better be preceded by a message to set attitude control to that special mode.
  During the pilot study this approach was applied to check a representative sampling of safety conditions against dozens of log files, each containing thousands of messages. However, safety conditions were checked one-by-one, permitting an approach in which the vast majority of a log file's messages could be ignored, so the amount of data put into the AP5 database remained relatively small.

2.3. Conclusions drawn from the first pilot study

Overall, the pilot study answered affirmatively its original question - *could simple analyses of spacecraft design information be performed rapidly by using a database as the underlying reasoning engine?*

- The database could readily be used to represent existing design information, and populating the database with that information could be accomplished rapidly by the judicious mix of manual effort and custom automation.
- Database queries could be used to perform simple analyses. The creation of these queries was a relatively straightforward, albeit manual, task.
- The efficiency of the database was sufficient for the volume of information dealt with in these pilot studies. However, questions remained about the scalability of the approach. In particular, checking properties of very large log files was anticipated to require a more efficient encoding of those properties. A state-machine based approach, e.g., (Andrews, 1998) or (Dillon & Yu, 1994) would perhaps be more appropriate in such circumstances.

For further details of this pilot study, see (Feather, 1998).

3. Verification and Validation of an Autonomous Spacecraft Planner

The need arose to perform V&V of an autonomous spacecraft control system. The following subsections provide background on the validation problem for the autonomous spacecraft control software, specifics of the spacecraft planner, and an overall description of the V&V approach followed. Later sections will describe in detail the development of the automated test oracle itself and its application.

3.1. Validating An On-board Planner for an Autonomous Spacecraft

NASA's "New Millennium" series of spacecraft is intended to evaluate promising new technologies and instruments. The first of these, "Deep Space 1" (DS1, 1998), was launched in 1998. Spacecraft autonomy is one of several innovative technologies that DS-1 demonstrated (NMP, 1999). The "Remote Agent" (Pell et al., 1996), (Pell et al., 1997) is the first artificial intelligence-based autonomy architecture to reside in the flight processor of a spacecraft and control it for several days without ground intervention. The Remote Agent achieved its high level of autonomy by using a software architecture with three key modules:

- an integrated planning and scheduling system that generated sequences of actions (plans) from high-level goals,
- a intelligent executive that carried out those actions and responded to execution time anomalies, and
- a model-based identification and recovery system that identified faults and suggested repair strategies.

The planner was a critical component of the autonomy architecture, and the one with which this paper is concerned. It

took as input a set of high-level goals, as provided by the ground operations team and the on-board navigation software, and generated a course of action (plan) that achieves those goals while satisfying various operations constraints. Validating the planner was of utmost importance. The command sequences generated by the planner direct navigation, attitude control, power allocation, etc. The entire mission could have been jeopardized by an error in a command sequence pertaining to any of these areas.

3.2. Specifics of The Remote Agent Planner

The planner generates a course of action (plan) that achieves goals provided by the ground operations team and onboard software modules and also obeys operations constraints as encoded in a declarative *domain model.* A plan consists of several parallel *timelines*, each comprised of a sequence of *tokens.* A timeline describes the future evolution of a single component of the spacecraft's state vector. Tokens can represent executable activities (e.g., thrust main engine), the state of resources (power, battery state of charge), and spacecraft state (e.g., attitude). For example, there may be one timeline describing the state of the engine (warming up, firing, or idle) and another describing the spacecraft attitude (e.g., pointing to a target, turning from target A to target B). Each token has a time interval during which it can start, a duration interval, and zero or more parameters. The token start times are synchronized by explicit temporal constraints in the plan. These serve to coordinate activity tokens as well as associate activity tokens with relevant resource and state tokens. For example in the plan fragment of Figure 3 the *thrust(B)* activity token must be contained by the *point(B)* state token—that is, the spacecraft must maintain the specified attitude while the engine is firing.

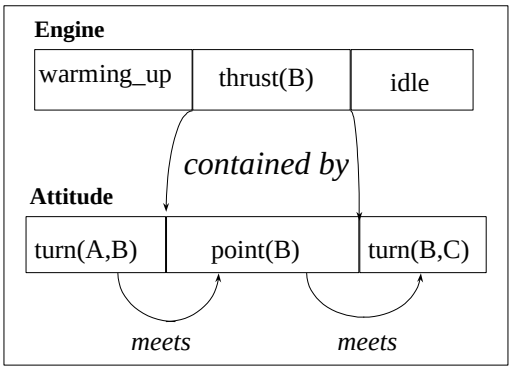


Figure 3: Plan Fragment

The plan must operate the spacecraft in a manner consistent with the accepted operations knowledge. This knowledge is encoded as a set of constraints in a declarative *domain model*. These express temporal relationships and parameter value constraints that must hold among the tokens in the plan. For example, a constraint may state that while the main engine is firing it consumes 1500 watts of power and the spacecraft attitude must point the engine axis along the desired thrust vector. Another constraint may state that camera images cannot be taken while the main engine is firing because the engine causes too much vibration.

```
((MICAS_Ready)
   :compatibilities
   (AND
    (met_by (MICAS_Turning_On))
    (meets  (MICAS_Turning_Off))
    (equal  (REQUEST (Power 15)))
```

Figure 4: A Compatibility Tree

More formally, the domain model is a set of *compatibilities*, each of which is a Boolean and/or tree of temporal relations that must hold between a *master token* and specified *target tokens w*henever the master token appears in the plan. If the master token does not occur in the plan, the relation does not need to be satisfied. An example is shown in Figure 4. This compatibility says that the *ready* state of the MICAS camera consumes 15 watts of power, and that the camera must be turning on just before it is ready, and turned off immediately afterward.

Whenever a `MICAS_Ready` token appears in the plan, it must be preceded by a `MICAS_Turning_On` token and followed by a `MICAS_Turning_Off` token, and the power timeline is decremented by fifteen watts for the duration of the token (and must not go below zero available power for obvious reasons).

The planner generates a plan that achieves a specified set of goals and satisfies the compatibilities in the domain model. The planner consists of a general-purpose reasoning engine and an application-specific domain model. It begins with a partial plan that contains only the initial spacecraft state and goals. It then employs a chronological backtracking search to find a complete plan that satisfies the compatibilities in the domain model. In each step of the search the engine determines which of the compatibilities in the domain model are not satisfied by the current plan. It selects one of these and determines which of a fixed set of plan modification operations would result in a plan that satisfies that compatibility. The planner selects one of those operations, applies it, and repeats the process on the resulting plan. If there is no operator that satisfies the compatibility, the algorithm backtracks. The planner architecture is discussed in more detail in (Jonsson et al., 2000).

3.3 Overall approach to V&V of the planner

The objectives and approach to planner verification and validation (Smith *et al.*, 2000) were as follows. The objectives were to verify that the domain model correctly encoded the operations knowledge acquired from the spacecraft engineers, and to verify that the planner enforced that knowledge—that is, the output plans achieved the goals from the initial state and satisfied all of the operations constraints encoded in the domain model. First, the operations domain knowledge was captured as a set of requirements in English and validated by the relevant spacecraft engineers in a series of review meetings. These requirements were then encoded in the domain model. To verify the planner, the planner was run on a number of test cases that were selected to thoroughly exercise the domain model. The resulting plan for each case was analyzed to (1) verify that it satisfied the constraints in the domain model and (2) verify that it satisfied the original requirements.

To fully exercise the planner required a test suite with hundreds of cases. Unfortunately, determining plan correctness is a time and knowledge intensive process. Analyzing these by hand would have been prohibitively expensive and error-prone. Some kind of automated test oracle was clearly needed, and the success of the first pilot study suggested the following approach:

- Automatically convert a plan into an AP5 fact database
- Check the correctness of this plan against the constraints input to the planner by expressing those constraints as AP5 queries. If the plan satisfied all of the queries, then the plan would be known to be consistent with respect to all the constraints.
- Check the correctness of this plan against the English-language requirements by expressing those requirements as AP5 queries. If the plan satisfied all of the queries, then the plan would be known to satisfy all those requirements.

The sections that follow describe the steps we took to estimate feasibility of this approach (section 4), development of the test oracle to check adherence of plans to constraints, including automating the translation of planner constraints into AP5 queries (section 5), extension of this oracle to check the English language requirements (section 6), and results of applying the complete oracle (section 7).

4. Second pilot study: feasibility study of V&V of an autonomous planner

The rapid analysis approach of the first pilot study was identified as having *potential* application to V&V of DS-1's planner. However, the first pilot study had examined traditional design information (interface diagrams and test logs), so there was uncertainty as to whether the same approach would work for the planner's inputs (i.e., goals, initial conditions and constraints) and output (i.e., plans). A second concern was motivated by the critical resource of planner experts' time. The first author, who was not a planner expert, had done the V&V research. Development of an automated plan checker would clearly require some investment of time by the planner experts - but how much?

A second pilot study was organized to ascertain feasibility of this approach, by seeking answers to the following two questions:

1) Could the database-based analysis approach be rapidly applied to automate checking the planner's generated plans against its temporal constraints?

2) Could this be done without a large investment of time by planner experts?

The area of most significant concern was that of scalability – would the AP5 database-based approach scale to handle DS-1's plans? We adopted a hybrid strategy to find the answer to this question: we used realistically sized plans, checked them against sampling of temporal constraints, and relied upon the generality of the checking language as compared to the planner constraint language to extrapolate the results. This point is discussed further in subsection 4.3. We entered into this pilot study with expectations of success, based on the recognition of the relative computational simplicity of checking vs.

planning. This, an instance of Blum's notion of "simple checker" (Wasserman & Blum, 1997), suggested that the development of a sufficiently efficient checker would not itself become a large development effort.

4.1. Architecture of the second pilot study

Figure 5 shows the architecture of the approach followed in this second pilot study. As before, it is organized into four main stages:

1. Creation of database schema to represent the plan's activities. This was confirmed to be a straightforward, manual task.
2. Loading the database with plan activities. This was made a completely automatic step in this pilot study. The amount of effort to do this was small, in part because both planner and database happened to be implemented in the same programming language (Common Lisp). Had there not been this fortuitous coincidence of a common implementation language, it would have been necessary to develop code to parse and translate between linguistic forms. At worst, this would have been a modest standard programming task.
3. Translation of constraints. Representative planner constraints were selected for hand-translation into the equivalent database queries. The study revealed translation to be feasible, although a somewhat detailed process (see the examples in the next subsection).
4. Analysis. As before, analysis was automatic, yielding reports of confirmations and anomalies. Importantly, this study confirmed that the database approach scaled sufficiently well to efficiently analyze representative plans. (The study used actual plans produced during test runs of the DS-1 planner.)

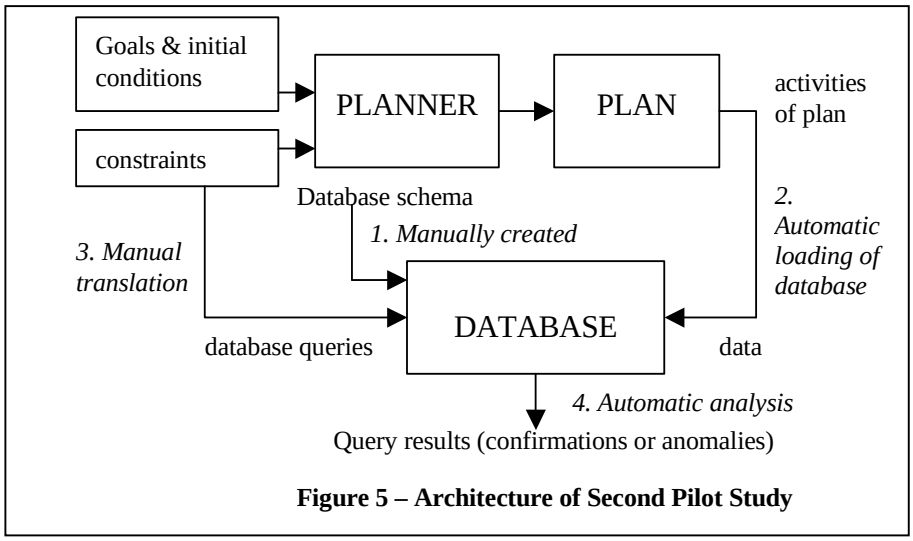


Figure 5 – Architecture of Second Pilot Study

4.2. Detailed Examples

4.2.1 Example of planner constraint

The following example of one of the simpler plan constraints, as expressed in the planner's special purpose language, will convey a feel for the challenges faced in this pilot study:

```
(Define_Compatibility
   ;; Idle_Segment
   (SINGLE ((SEP_Schedule SEP_Schedule_SV)) (Idle_Segment))
   :duration_bounds [1 _plus_infinity_]
   :compatibility_spec
   (AND
    ;; Thrust and Idle segments must all meet--no gaps
    (meets
     (SINGLE ((SEP_Schedule SEP_Schedule_SV))
             (Thrust_Segment (?_any_value_ ?_any_value_)))))
    (met_by (SINGLE ((SEP_Schedule SEP_Schedule_SV))
                    ((Thrust_Segment (?_any_value_ ?_any_value_)))))))
```

This illustrates several areas where knowledge held by the planner experts had to be acquired by the V&V expert:

- **Overall application domain knowledge:** "`SEP`" is an acronym for "Solar Electric Propulsion," the innovative engine that provides thrust to DS-1. "`Thrust`" and "`Idle`" are the two main states this engine can be in.

Knowledge such as this of the spacecraft domain provided useful intuition to the V&V expert, and this second pilot study warranted a deeper level of understanding than had been necessary for the first pilot study.

- **Problem-specific terminology:** "`SINGLE`" has a connotation specific to DS-1's planner. It introduces a description that matches a single interval. (One alternative is "`MULTIPLE`," introducing a description that matches a contiguous sequence of intervals).
- **Terminological variants:** The overall definition is that of a "compatibility." The V&V expert preferred to think of this as a "constraint," in keeping with the terminology of the database tool. Another example is the "`?_any_value`" term, which serves as a wildcard, indicating any acceptable parameter value may occur in the corresponding parameter position. Again, the V&V expert had the exact same concept, but preferred a different syntax.
- **Confirmation of shared understanding:** there were some areas of shared understanding, but these had to be confirmed, not taken for granted. A trivial example is "`AND`", which in the above is used to indicate that the constraint (compatibility) holds if all of the clauses of this `AND` hold. More interesting are the terms "`meets`" and "`met-by`," which are binary temporal relations between intervals, drawn from the work by (Allen, 1983).

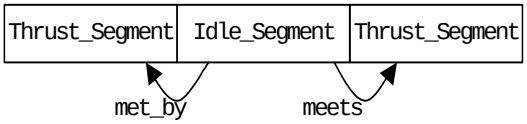
The net result was that the V&V expert required an intensive session of coaching on the meaning of the planner notations (plans and constraint language) at the start of this pilot study, and incremental assistance at various points throughout. Overall this did not amount to an undue consumption of planner experts' time.

4.2.2. Example of Translation from Planner Constraint to Database Query

Consider the `Idle_Segment` constraint given earlier. Its essential core is the following:

```
(SINGLE ((SEP_Schedule ...(Idle_Segment))
:compatibility_spec
(AND
 (meets (SINGLE ((SEP_Schedule ... (Thrust_Segment (?,?)))
 (met_by (SINGLE ((SEP_Schedule ... (Thrust_Segment (?,?))))
```

The fragments `(SINGLE ((SEP_Schedule ...` introduce descriptions that are to match to activities of the SEP scheduled in the plan. The first such description is of an `Idle_Segment` activity. For every instance of an activity in the plan matching that description, the constraint requires that the logical condition `(AND ...)` is true. The logical condition is the conjunct of two clauses. The first says that the matching instance `meets` a `Thrust_Segment` activity, i.e., the end-point of the `Idle_Segment` activity exactly coincides with the start point of some `Thrust_Segment` also in the plan. The second says that the matching instance is `met_by` a `Thrust_Segment` activity, i.e., the start point of the former exactly coincides with the end point of the latter. Pictorially:



For translation, this is split into two separate constraints, one for each clause of the conjunct. This allows the checking to be conducted separately for each conjunct, so that any anomaly in a plan can be narrowed down as much as possible. The translated form of the first such conjunct looks close to the following (it has been tidied up slightly for presentation purposes):

```
(A (x) (IMPLIES
    (activity-in-plan x Idle_Segment SINGLE SEP_Schedule)
    (E (y) (AND (activity-in-plan Thrust_Segment SINGLE SEP_Schedule)
                (meets x y)))))
```

`A` and `E` are the database's notations for the logical concepts for-all and exists. `IMPLIES` and `AND` have the standard logical meaning. `activity-in-plan` is a ternary relation (defined for plan checking) that relates an activity name (e.g., `Thrust_Segment`) to a keyword (e.g., `SINGLE`) and schedule (e.g., `SEP_Schedule`). `meets` is a binary relation (again, defined for plan checking) that relates two activities if and only if the end point of the first coincides exactly with the start point of the second.

For this pilot study, some of the more complex planner constraints were also selected for hand-translation. Their additional complexity stemmed from references to activities' parameter values. For example, a constraint that says that every `Max_Thrust_Time` interval whose 1st parameter is `100` must end an `Accumulated_Thrust_Time` interval whose parameters are respectively `100`, `0`, the same value as `Max_Thrust_Time` interval's 2nd parameter, and `WHILE_NOT_THRUSTING`.

4.3. Conclusions drawn from second pilot study

The study answered affirmatively its first question - *could the database-based analysis approach be rapidly applied to*

automate checking the planner's generated plans against its temporal constraints?

It demonstrated the feasibility of automating checking of plans. This was predicted to be an onerous task were it to be done manually, for the following reasons:

- DS-1 plans are detailed and voluminous, ranging from 1,000 lines to 5,000 lines long. They are designed to serve as input to DS-1's remote agent smart executive, another automatic component, rather than to be perused by humans. In particular, determining whether a temporal constraint is satisfied by a plan requires searching for information that is dispersed throughout the plan.
- The DS-1 planner spacecraft model has several hundred temporal constraints.
- During testing of DS-1's planner, thousands of plans would be generated.

Furthermore, automatically checking all the temporal constraints of all the generated plans was recognized to be desirable. Manually checking of plans would permit only a small fraction of these generated plans to be scrutinized in detail, so would necessitate the (challenging) selection of which to scrutinize. In contrast, automating the checking of plans against all the temporal constraints would permit *all* the generated plans to be so checked, thus giving confidence in the correct operation of the planner.

The extrapolation from success on the pilot study's examples to a prediction of success on the full problem relied upon the following observations:

- The pilot study used realistically sized plans. At this state of development, an early-version DS-1 planner model of the spacecraft existed, and could be used to generate plans. The assumption was that while the details of this planner model would change (e.g., as precise characteristics of the hardware it was modeling become better known), and new temporal constraints would be added, the overall *size* of the plans would not change drastically. This assumption was based on knowledge that (1) the plans would cover times period of predetermined fixed length, (2) the plan steps, corresponding to the operations of spacecraft hardware, would stay at the same level of granularity, and (3) the plan complexity would not increase drastically, for the simple reason that the DS-1 spacecraft hardware's execution of these early plan models was already close to its resource limits.
- The database query language is less restrictive than the planner language. This is because the planner has to be able to generate plans; its constraint language is crafted to simultaneously ease the expression of certain constraints, and limit the form of expression to those that it can readily handle. Conversely, the database only has to be able to evaluate queries about a specific set of data, a far easier task than the search-intensive task of planning. The database query language is an extensible, general-purpose language and so should be capable of straightforwardly expressing the planner's constraints. Hence the successful encoding a sampling of temporal constraints as database queries suggested that encoding all the temporal constraints would be possible.

The second question - *could this be done without a large investment of time by planner experts?* - was also answered affirmatively. The planner experts spent some time to bring the V&V expert up to speed, and thereafter to answer occasional questions. Their total time expenditure was not excessive. Interestingly, while the amount of time expended by planner experts on this task remained well below that expended by V&V expert, it was noticeably higher than had been the case for the first pilot study. Generally, we attributed this to the need to delve into more application-specific details, resulting in the need for more coaching of the V&V tool expert by the spacecraft planner experts. The first pilot study had looked at relatively generic design information, the expression and meaning of which was fairly standard. The planner's inputs and outputs were expressed in a planner specific notation, the semantics of which were not intuitively obvious.

The success of this pilot study led to the commitment to construct a full oracle and use it as a regular part of the planner test program, discussed in the next two sections.

5. Development of the Oracle for Planner Constraints

The success of the second pilot study led to the next phase – a commitment to develop an automated oracle to check the correctness of plans. The oracle would be used during testing of the planner by the planner experts themselves. While this might appear to be just a small extension of the previous phase, there were several important ramifications of this transition from pilot study to actual development:

- **Reliance upon the result:** The pilot shadowed the actual spacecraft development effort, but did not promise to yield results upon which that development effort would rely. Indeed, a valid result of the pilot study could have been that the approach was infeasible. In contrast, this phase committed to the development of a tool that the project would rely upon during testing.

  The positive results of the pilot studies were necessary precursors to this commitment. Additionally, our realization that the analyzer employed an extensible, general-purpose language gave us a justification of why we could extrapolate those

positive results to the entire planner constraint language.

- **Developer and end-user different people:** The pilot study tools were developed primarily by the V&V expert, and used by that same person. In contrast, this phase committed to the development of a tool that would be applied by the planner experts with little, if any, involvement of the V&V expert during use.
  This motivated two extensions to the approach demonstrated in the second pilot study: (i) automating the translation from planner constraints into database queries, and (ii) rendering the outputs of the analysis step in terms understandable by the planning experts.
- **End-user agenda:** the DS-1 planner experts constructed an agenda of capabilities they desired of the to-be-developed tool. This featured a prioritized list of capabilities, such that the capabilities to be developed sooner would be the ones they predicted would be of more value to them.
  The preceding pilot studies had helped by providing illustrations of the kinds of analyses that could be accomplished employing this approach. The fact that those illustrations were in terms of DS-1 specific information contributed to their (the planner experts) ability to see its potential. They were thus able to formulate an agenda at this stage, supplanting what was previously the V&V tool expert's *guess* as to what analyses might be interesting and/or valuable.

The architecture of the system developed in this phase is shown in Figure 6. For the remainder of this paper we will refer to this system as the "planchecker". It has the same stages as the second pilot study, but with some additional capabilities:

- **Additional analyses:** the planner experts asked for further analyses beyond temporal constraints, notably type checking of plan elements, and cross-checking of plan activities against their rationale (information on which is included in the generated plans). These required loading additional information from plans into the database, and development of additional database queries.
- **Automatic translation:** there were over 200 instances of temporal planner constraints (counting each lowest-level clause as one constraint). Based on the observations of the second pilot study, we recognized that manual translation of the whole set would be a tedious task. Worse yet, we expected the set of planner constraints to grow and change over time, as successive releases of the planner model would be created in response to bug fixes. In keeping with our overall goal of judicious use of automation, it was decided build an automatic translator that would take *any* constraint expressible in the planner language and generate the equivalent database query. The planner constraints fell into a small number of categories, so we judged that a general translator could be constructed.
- **Extended output:** the planner experts wanted the query results to report more than simply "OK" when a plan passed the checks. In essence, they wanted a justification for *why* a temporal constraint was satisfied. For example, a constraint that says every engine-thrusting interval is followed by an engine-idle interval would be justified by listing, for each engine-

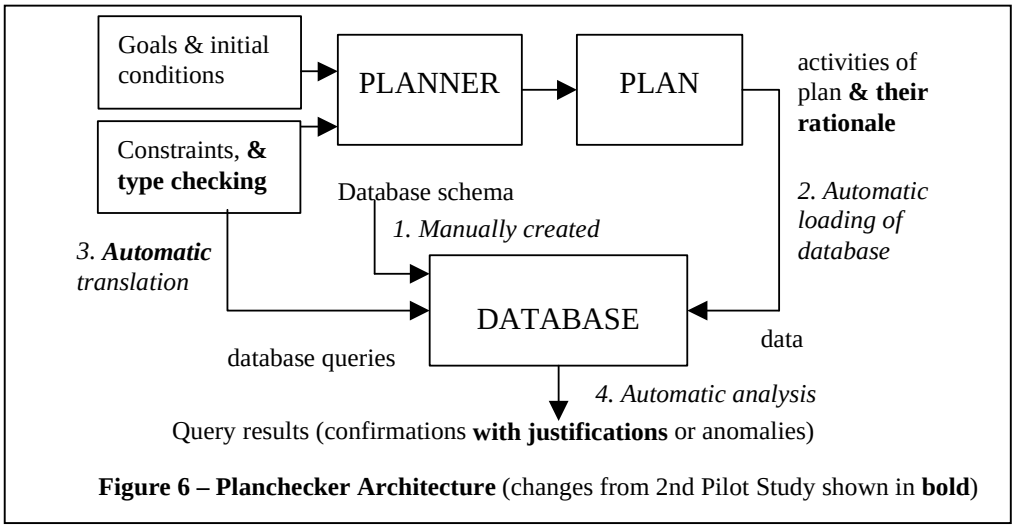


Figure 6 – Planchecker Architecture (changes from 2nd Pilot Study shown in **bold**)

thrusting interval, the specific engine-idle interval found to satisfy the constraint. The need for this extended output was twofold:

- Extrapolate from the success of the planner at generating a specific plan to gain confidence about the planner in general, and
- Make available the information that the planner experts would need to guide them in debugging erroneous plans. This was especially important. For the same reason that a plan is laborious to check manually, it is laborious to debug manually. The planchecker's justifications help guide the planner experts to the locations in the plan related to a particular constraint.

- **Coverage analysis:** the planner experts also wanted to know *which* of the planner constraints had been exercised in the plan. For example, only plans that involved engine-thrusting intervals would exercise a constraint of the form "every engine-thrusting interval must …".

5.1. Detailed Examples

5.1.1. Automating the translation from planner constraints to database queries

The hallmark of this task was the need to deal with many small (and to the V&V tool expert often surprising) details. Most commonly, these were details of the plan constraint language that the V&V tool expert had not encountered earlier. The representative sample of constraints hand-translated in the second pilot study did not cover the full range of constraint language constructs. The discovery of these came to light when the partially developed planchecker was applied to increasingly more of the entire set of DS-1 constraints, and to increasingly many of the plans that had been generated. They manifested themselves in one of three ways:

- **Error (break) during translation, loading or analysis.** For example, if the constraint translator encountered a variable in a location where it expected a constant. Generally, these were easy to find and understand. A break in the middle of analysis required some simple debugging-like activity to trace back to the underlying discrepancy. Since the database was implemented on top of Common Lisp, the power of the run-time environment available in the middle of a break made this task fairly simple.

  All these cases resulted in a simple question that the V&V expert would ask of the spacecraft planning experts (e.g., "what does it mean to use a variable name as a range value where normally there is an explicit integer?")
- **False alarms - spurious anomalies detected by analysis**. Often the automated steps would complete, but would report a whole host of (as it turned out, spurious) anomalies. The V&V tool expert generally interpreted a large number of anomalies to be indicative of a flaw in his understanding, rather than a grossly incorrect plan. Indeed, genuine plan anomalies were so few and far between that this was an effective working hypothesis.

  The crucial issue in these cases was finding the underlying cause of the spurious anomalies. The V&V expert would spend time to narrow down the likely cause of a reported anomaly. This culminated in a question to ask of the spacecraft planning experts. For example, suppose this was the first analysis of a plan that exercised default interval range values for one of the temporal relationships. An "anomaly" that could be traced back to one of these defaults would be indicative of a misinterpretation of what the default should be. The V&V expert would then know to ask a specific question about that default value.

  This was a somewhat labor-intensive process for the V&V tool expert. Its benefit was that it ensured that the planner experts' (very limited) time was not squandered unnecessarily.
- **False approval – failure to detect anomalies.** The surprises that were hardest to recognize and understand were those concerning failure to detect anomalies. The redundancy of the information in plans was especially useful to help detect these cases. See V&V lesson 1 (in section 8) for discussion of this issue. Additionally, the V&V tool expert followed the traditional approach of seeding genuine plans with deliberate errors, and observing whether the analysis caught them.

5.1.2. Structuring analysis results

The need to structure analysis results to be more than simply "pass" / "fail" was a strong theme of the planchecker development. Some examples of the need for this are as follows:

- All the DS-1 planner constraints take the overall form: for every activity-1 that matches description-1 there exists an activity-2 that matches description-2. A constraint of this form is *trivially* satisfied if the plan contains no activities matching description-1. The planchecker separates trivial and non-trivial cases in its reports of constraint satisfaction.
- The DS-1 planner generates plans for a segment of the entire mission (e.g., one week). Thus a plan is bounded within some "horizon"– it has a start and an end. Yet, the constraints may extend across this planning horizon. Such an instance is reported as a special kind of constraint satisfaction in which the plan satisfies the constraint within its horizon, but defers some residual checking for the next plan. The details of all such deferred checks are included within the planchecker's report.
- In an early version of the planner, a few of the constraints referenced information that is not stored in plans. In essence, this external information directed which one of several constraints is to apply. The planchecker's constraint translations handle these circumstances by checking each alternative. If all fail, it is an anomaly. If the plan is found to satisfy one of the alternatives, again, a special kind of constraint satisfaction is reported, which included the deduction of what the external information must be to direct the choice of the satisfied constraint.

The details are domain-specific, but we see a recurring need to make distinctions among classes of "pass" reports, and structure the analysis results accordingly.

5.2. Insights gained from development experience

The development effort did indeed culminate in the planchecker oracle (use of which is discussed in the next section). We therefore confirmed the validity of the conclusions drawn from the second pilot study. We also gained some further insights. These fell into two key areas:

- The second pilot study had suggested that the translation from planner constraints to database queries would be straightforward. In practice, automating the translation of the full planner language turned out to be more complex than the pilot study had indicated. While a procedural approach to programming the planchecker's translator sufficed to meet the development goals, we concluded that translation warrants further attention. We will return to this in Section 8, Lessons Learned.
- In practice, testers need analysis results with more content and structure than simply "pass" or "fail". Further discussion is deferred to Section 8, Lessons Learned.

6. Extension of the Planchecker Oracle to Validate Requirements

The second author (a spacecraft planner expert) extended the planchecker to check the English-language planner requirements. Motivation for this checking stems from limited forms of expression allowed in the planner constraint language. The limitations on forms of expression were there to make planning itself tractable. As a result, the writers of planner constraints had often found it necessary to manually decompose operations knowledge acquired from the spacecraft engineers and captured as requirements written in English, into a *set* of constraints that the planner would accept, and that in combination would achieve the original constraint.

Because the database query language was not so tightly constrained, it was often possible for the planner expert to express the original requirements as a *single* database query. This could then be applied to automatically check plans. Figure 7 shows the architecture of this extended use of the planchecker.

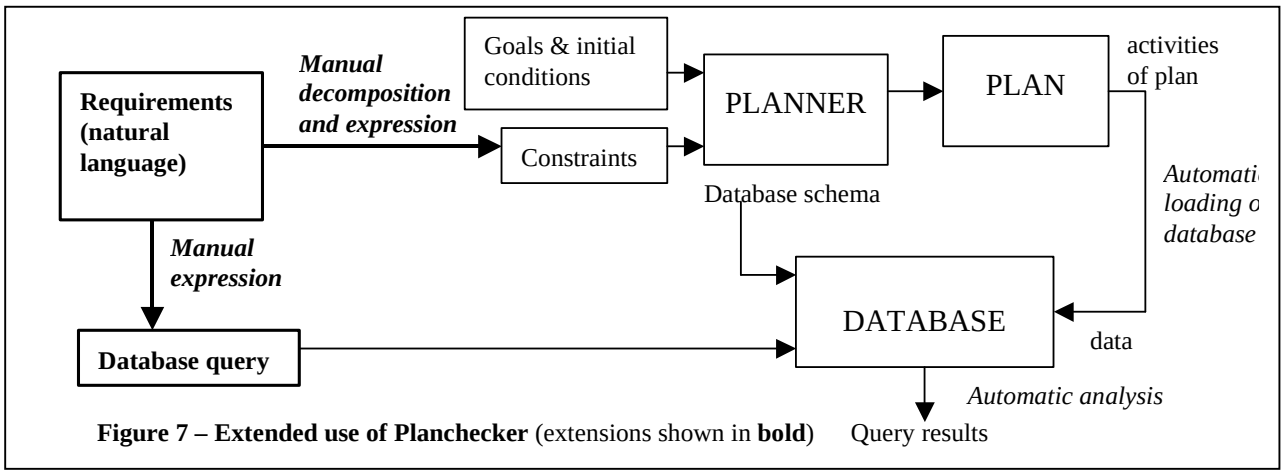


Figure 7 – Extended use of Planchecker (extensions shown in **bold**)

In essence, this provided an important element of *validation*. The planning expert was able to confirm that the set of constraints, expressed in the planner's input language and used by the planner itself to generate plans, indeed achieved the requirement that was originally envisioned. The key to this was the availability of both a general-purpose language in which the requirement could be expressed straightforwardly, and the automatic capability to translate such expressions into their corresponding test oracles.

Note that prior to the existence of the planchecker, there was no incentive for planner experts to use a formal language to state their requirements, since no general method existed to automatically convert such constraints into an equivalent set of constraints in the limited expressivity of the planner language. Once the planchecker existed, the benefit of test oracles motivated the formal expression of those requirement. Of course, the planner experts still had to manually convert them into planner constraints, but now they had the additional assurance of the validity of that translation on all the test cases they executed and checked.

The implications of this are:

- A planner expert was able to master the use of the database language and the special-purpose constructs added to represent and reason about plans. Seeing familiar examples (translations of the standard constraints) helped in achieving this level of understanding.
- The planchecker architecture facilitates such extensions – specifically, automatic loading of plans into the database,

and automatic evaluation of database queries, can both be reused. (Of course, the translator from planner constraint language could not be reused, because the original constraints were not expressible in that language.) The net result is extra validation at the cost of very little extra time and effort.

- The gap between requirements and test can be made narrower than the gap between requirements and generation. This is key to effective validation.

7. Use of the Planchecker Oracle

The planchecker was used by the second author (a planning expert) during testing. Interaction with the V&V expert was not required during this phase. The full planchecker was run on each release of the Remote Agent software. There were a total of nine releases occurring six to eight weeks apart. This testing process discovered and fixed over 200 bugs, and the planner performed flawlessly during the experiment on board the spacecraft.

In use, the planchecker's results were accumulated alongside other statistics about the plan generation, e.g., how long it took to generate the plan, and how much memory was required to do so. It was easy to apply in "batch mode" to a whole series of plans. It was tolerably efficient, taking times in the range of 30 seconds to 4 minutes to complete the checking of a typical plan. In comparison, generation of these plans took from 3 to 10 minutes, so planchecking was never a bottleneck. (Note that these are timings taken during testing, when running on fast ground-based hardware. In operation, the planner took longer when running on the limited hardware resources on the spacecraft itself.)

Over the course of use, several sets of changes were made to the planner constraints. Re-translating the entire set of constraints, to generate a new instance of the planchecker oracle, easily accommodated these changes. On these occasions the V&V tool expert was on hand. The re-translations went smoothly, with only one instance of the need to step in and make a corrective modification. There were even changes to the plan format, in response to which the V&V tool expert had to (manually) adjust the corresponding portions of the planchecker system.

7.1. Impact of the Automated Test Oracle

To fully exercise the planner required a test suite with hundreds of cases. Analyzing these by hand would have been prohibitively expensive and error-prone. The plans are several hundred kilobytes long and intended for computer, not human, readability. The domain model consisted of over 200 individual constraints, and 48 requirements. Checking each one manually would have required several hours, with high likelihood of error. The automated plan analysis capability was clearly useful.

The utility of automatic domain model conversion was also proven. As shown in Table 1, at least 33% of the temporal relations, which comprise the compatibilities, changed over the nine releases. The requirements did not need automatic conversion since they were expected to remain stable. Additionally, the plan checker provided information from which to determine coverage analysis. This was useful information for assessing how well the test cases exercised the domain model.

The full oracle successfully detected three broad classes of defects as summarized in Table 2: violations of high-level requirements, failure to enforce the domain model compatibilities, and syntax errors in the domain model and plan. The oracle detected 84 of the 129 defects detected by formal testing. The remaining defects were detected by a trivial plan convergence test (did the planner generate a plan within stated time limits). Most of these defects were violations of the high-level requirements—that is, the model failed to properly encode the operations knowledge. The oracle detected a handful of syntax errors in the domain model and plan output, and there were no defects in the planner's enforcement of the domain

Table 1: Model Size by Release

Version	timelines	tokens	token parameters	compatibilities	temporal relations
000	18	36	51	44	153
008	19	38	50	43	156
011	18	37	50	40	178
015	18	37	55	39	175
019	18	37	55	37	175
026	18	37	58	39	180
027	18	37	58	41	198
029	18	37	59	46	212
Flight	18	37	59	46	211

model constraints. This last result was actually consistent with our initial expectations. The plan engine had been used for a

number of other research projects, and was believed to have few defects in this area. However, it was imperative to check this assumption formally in order to flight qualify the planner. The oracle *did* detect a handful of syntax errors in the domain model and output plan, which were not expected. Many of these were of low severity, but a couple of the plan syntax defects could have impacted plan execution under certain circumstances and were therefore of higher severity.

Overall, we judge this approach to developing an automated test oracle to have succeeded, and contributed significantly to the validation of the DS1 Remote Agent Planner. The development costs were a small fraction of the overall test effort, and it produced huge savings in test analysis costs. In fact, without an automated test oracle it would have been impossible to validate the DS1 planner in a cost-effective manner. Similar oracles based on this technology are being considered for validating other planning systems at JPL.

Table 2: Oracle-detected Defects by Class

Defect class	Defects
Constraint not enforced	0
Model/plan syntax error	14
Requirement Violation	70
Total	84

8. Lessons learned

The lessons we draw from this experience are presented next, beginning with those related to general software engineering principles, followed by those specific to V&V. For each, we detail our specific experiences in developing and using the planchecker, and then go on to discuss the more general issues of applicability to domains beyond testing of planning systems.

8.1 Software Engineering Lesson 1: Pilot Studies

Our experience re-iterates several well-understood virtues of pilot studies as a precursor to actual development.

Pilot studies provide evidence of feasibility, serve as prototypes and yield examples, which inspire suggestions for extensions, further applications, etc. Executable prototypes can demonstrate acceptable run-time efficiency (or lack thereof).

In addition, we found it useful to formulate a justification of why the pilot study approach would extend to the full problem. This was needed because we were seeking to apply a relatively novel combination of technology (use of a database to underpin a test oracle for a planning system). This meant that we had little in the way of past experience to serve as guide. Additionally, the cases we had considered in the pilot study did not necessarily cover all the features of the full planning constraint language. The justification we emerged with took advantage of the comparison between the restricted nature of the domain-specific planner constraint language and the general-purpose nature of the analysis language. This justification nicely complemented the evidence provided by the pilot studies' specific cases.

Note the power of domain specific notations (e.g., conciseness) comes from their suitability to purpose. Thus in proposing to substitute a general-purpose notation for a domain-specific notation incurs a potential expansion in going between the two notations. We circumvented this concern by automating the translation between notations (albeit at the cost of building that translator), so that the cost would be borne by automation, not by additional manual effort.

8.2 Software Engineering Lesson 2: "On-Demand" Knowledge Acquisition

When documentation is incomplete and domain experts' time is a critical resource, follow an "on-demand" policy of knowledge acquisition.

At the start of the project the V&V expert lacked a complete and fully documented specification of the task (i.e., plans and the planner language). Furthermore, the domain experts' time was very limited. In response, we followed an "on demand" approach to knowledge acquisition, where the V&V expert would proceed as far as possible before making the next enquiry of the planner experts. This made good use of the planner experts' limited time and availability, since it kept the sum total of their time small, consumed it in small chunks, and could be done asynchronously (e.g., via email exchanges, supplemented by brief telephone calls).

We benefited from the existence of numerous sample inputs (plans and planner constraints). Also, the nature of the task clearly circumscribed the areas that the analysis expert would have to master.

We found it useful to work from an example plan that a planner expert had already vetted as being correct. If the planchecker reported faults with such a plan, the V&V expert would know that most likely there was an error in his own

understanding, or his coding of the planchecker itself. Any remaining anomaly that the V&V expert could not resolve would then be a plausible candidate for a genuine plan anomaly, something the plan expert was very interested in!

Applicability: when is "on-demand" knowledge acquisition necessary and possible?

Lack of complete documentation is likely to be common in fast-paced development efforts conducted by closely-knit teams. In such situations, "on-demand" knowledge acquisition would be appropriate to bring a new team member, or adjunct to the team, up to speed.

Availability of examples of correct behavior can go a long way to providing information (such as in our case, where the adjunct was a V&V expert). Scenarios, for example those encouraged by UML practitioners (Douglass 1998), may contain the needed level of detail.

8.3 V&V Lesson 1: Encourage and Use Redundancy and Rationale

V&V can make good use of redundancy and rationale, to increase assurance in the V&V results, and to assist in the development of the V&V technology itself.

Each plan generated by the spacecraft planner contains both a schedule of activities, and a rationale relating those activities to the constraints taken into account in their planning. Checking both of these might appear redundant – surely what really matters is whether or not a plan satisfies all the constraints. Nevertheless, we found this redundancy to be useful in two ways:

1. The planner experts gained additional assurance that their generated plans were correct, in particular, that they generated the "right" results "for the right reasons."
2. The V&V tool expert made use of the redundancy to extend (and debug) his understanding of the task. Every constraint that the planchecker identified as being involved had to be identified in the plan's rationale, thus forcing the planchecker to be complete and correct in its treatment of rationales. Likewise, every constraint mentioned in the rationale had to be seen to be involved by the planchecker, thus forcing the planchecker to be complete and correct in its treatment of constraints. This helps assure that the planchecker is not reporting "false positives" (plans judged as correct which are actually incorrect). (Andrews, 1998) describes false positives as more serious than false negatives. He suggests, "…a thorough system of document reviews …can mitigate the risk of these false positives." Our experience indicates that machine-generated rationale can provide a basis for automating some of this review process.

Applicability: when can V&V feasibly employ redundancy and rationale information?

Not all domains will have interesting redundancy. For example, had our task been to validate the more traditional style of spacecraft control software, there would have been little, if any, redundancy or rationale information to exploit. Rationale in particular seems to be a hallmark of knowledge-based systems. Expert systems, planners, schedulers, and theorem provers are instances of systems that manipulate rules of inference so as to arrive at a result.

We observe that many knowledge-intensive systems offer some form of trace of the reasoning process they followed. For example, expert systems reveal a trace of their reasoning process that users can inspect to assure that they are indeed following accepted chains of reasoning to arrive at their conclusions. This was the case in our V&V of DS-1's planner – the rationale information was already being generated as part of the output. The way we made use of it is akin to the way a proof checker makes use of a theorem prover's proof-trace: such a proof-checker is easy to build and, relative to the cost of finding the proof in the first place, inexpensive to apply.

8.4 V&V Lesson 2: Database-based Analysis

The use of a database as the underlying analysis engine has practical applications and benefits.

Based on the first of our pilot studies we had made the argument that database-based analysis was suited to "lightweight" V&V (Feather, 1998). The success of this whole effort strengthens our belief in this position, and highlights some further benefits.

The database approach suggests a natural decomposition of the problem into: translating the V&V conditions into database queries, loading the data into the database, performing the analyses, and generating the reports. This simple architecture nicely separates the key steps. For example, in response to a change in format of plan structures it sufficed to modify the planchecker's database loading portion. Also, this architecture facilitated the planner experts' extended use of the planchecker (i.e., their checking of complex requirements by manually expressing them as database queries).

The database itself is used as intermediary between analysis and report generation steps. The planchecker places analysis results back into the database, alongside the original data (plans) from which those results are derived. Thus the report generation phase has uniform and simultaneous access to both kinds of data regardless of source, considerably facilitating the report generation task.

Applicability: what are the prerequisites for database-based analysis?

We identify the following as the key prerequisites, and go on to discuss their ramifications:

- Explicit data is available for analysis
- Data is accessible through files
- A "batch" style of analysis is acceptable
- V&V conditions are formulatable as database queries

The need for *explicit* data is exemplified by contrasting this database-based approach with model checking. We used database-based analysis in the DS-1 domain to analyze the *explicit* products of test runs, namely the plans generated by the planner. While we can apply the same approach to design information (e.g., to check that interface specifications of different components indeed match (Feather, 1998)), again, this must be explicit information. Conversely, model checking works with an *implicit* description of a state space, namely a state machine, and checks that properties hold of all the state trajectories implied by that machine. Model checking may take advantage of the implicit nature to arrive at its answer far faster than would be possible if the state space had been made explicit and the question asked of that explicit state space.

Accessibility of data though a file is less significant of a restriction, since most forms of test execution or simulation can be configured to yield log files. Analysis of timing requirements, for example, could be accomplished by having the log file record timestamps along with events.

By *batch* style of analysis we mean that the entire set of data to be analyzed be available before analysis commences. This precludes the application of our approach to on-the-fly analysis while a test run is taking place. On occasion such a capability can be very useful, say to terminate a test run that has already revealed a bug (and so whose behavior past that point is not of value), or to monitor an ongoing activity (so be able to invoke a fault protection mechanism if an error is detected).

Our experiences suggest that it is usually practical to formulate V&V conditions as database *queries*. As discussed earlier, we made use of an extensible and general-purpose query language (essentially first-order predicate logic). This offered expressive power ample for most of our needs.

At the more complex end of the spectrum of V&V conditions were the requirements discussed in Section 6. The equivalent AP5 queries were not necessarily elegant or simple, but generally matched the natural language formulation in a straightforward manner.

At the simpler end of the spectrum of V&V conditions expressivity was never in doubt. What mattered was the ease by which the automatic generation of queries could be accomplished. For example, in the transition from second pilot study to commitment to develop the planchecker, the planner experts asked that the V&V checking be extended to incorporate *type checking* of plans. This was a set of very simple checks:

- every instance of an interval in a plan is of the correct type (e.g., the portion of the plan dealing with solar electric propulsion must contain only intervals whose types are one of the enumerated solar electric propulsion types).
- every instance of an interval in a plan have the correct number of arguments with the correct types (e.g., the first argument of a solar electric propulsion standby interval be a non-negative integer).

The equivalent database queries were simple to express. Importantly, having established the core capability of oracle generation, accommodating further simple checks was a very low-cost extension. As we have seen from the loss of the Mars Climate Observer spacecraft (Mars Climate Orbiter, 1999), mistakes in items as simple as units can lead to disaster. An approach that quickly and easily accommodates checks for such mistakes has practical value.

8.5 V&V Lesson 3: Analysis Results Need Detail and Structure

Test oracles should yield results with far more content and structure than simply "passed" or "failed".

During the pilot studies it had sufficed to yield analysis results with trivial structure – they reported either that the object had "passed" the analysis test, or had "failed due to…." (with some simple distinctions among failure cases).

The planchecker development entailed the generation of analysis results and reports with considerably more structure to both the "passed" and "failed" cases. Specifically, reports identified *which* constraints had been exercised by a plan, and that distinguished *how* constraints had been satisfied: those that were wholly satisfied by the plan, those that deferred some condition to activities beyond the plan's horizons, etc.

For example, consider the constraint that required every interval of thrusting by the ion engine to be contained by an interval of constant pointing during which the spacecraft's solar panels were oriented towards the sun.

- For a plan that contained no intervals of thrusting, the report would indicate that this constraint was "trivially satisfied".
- For a plan that contained one or more intervals of thrusting, the report would indicate for each such interval whether or not the constraint was satisfied. If satisfied, the constant pointing interval would be listed alongside the thrusting interval; if not satisfied, the plan would prominently highlight this failure, and list the thrusting interval(s) for which no containing constant pointing intervals could be found.

We suspect that there may be general principles by which test oracles can be built to yield such structured analysis results,

an area we think is worthy of further attention.

Applicability: when do details of analysis results need to be presented, and what should their structure be?
We see the following as motivating factors:

- Test Partitioning
- Dispatching for further treatment
- Debugging

Testing can rarely cover more than a small fraction of possible system behaviors. Partition testing is one way to approach testing of complex systems (see (Gutjahr, 1999) for a recent discussion of partition testing and its effectiveness). Test results' details can indicate where those test cases lie within the space of possible tests, and so be of use to guide further test selection. A simple example is testing an assertion of the form *A implies B*. This is trivially satisfied by any test case in which *A* is false. More revealing is a test case where *A* is true (and, in order to pass, *B* is true). The example cited above, that every interval of thrusting be contained by…, fits this pattern.

Some analysis results may warrant further treatment, depending upon their details. For example, a plan that deferred some condition beyond that plan's horizon might warrant further checking in conjunction with the preceding and/or following plan.

When analysis reveals a problem, it is obviously helpful to provide details of the problem to aid debugging.

8.6 V&V Lesson 4: Translation is the key

Translation between notations is a recurring need, and ideally should be done in such a way as to support understanding, specification and maintenance by domain experts.

The planchecker, and the pilot studies that preceded it, made extensive use of translation between notations. For example, the loading of a plan into the database was a simple translation from plan format into database schema format.

In the pilot studies, it sufficed to perform these translations manually, or to develop procedural-style code to automate the translation. In development of the planchecker, translation from planner constraint language to database query language was also programmed procedurally, but, because of the complexity of this translation, this had some untoward consequences. Notably, the procedural code was hard to understand and maintain.

We believe that for translation of this complexity, a more declarative style would be superior. In one such approach, translation would be expressed as a set of translation rules, executed by a general-purpose translation rule engine. We would hope that such translation rules are readily created, understood and maintained. Subsequent to the development and use of the planchecker, we have explored this issue by constructing a grammar for the entire DS-1 constraint language in POPART (Wile, 1997), a parser-generator tool. Figure 8 shows a fragment of the grammar, and a fragment of a DS-1 constraint that parses as a CompositeCompatibilitySpec in the grammar. With this grammar we are able to parse all the DS-1 constraint files.

```
CompatibilitySpec :=
    TemporalCompatibilitySpec | CompositeCompatibilitySpec ;

CompositeCompatibilitySpec := '( LogicalOp CompatibilitySpec + ') ;
LogicalOp := 'AND | 'OR ;

TemporalCompatibilitySpec := '( TemporalOp { TemporalBoundsSpec#1 }
                        { TemporalBoundsSpec#2 } BTokenSpec ') ;
TemporalOp := 'contained_by | 'contains | 'equal | 'meets | 'met_by |
          'starts | 'ends | 'before | 'after | 'starts_after |
          'ends_before | 'starts_before | 'ends_after ;
TemporalBoundsSpec := IntervalTemporalBoundsSpec |
                  VariableTemporalBoundsSpec ;
IntervalTemporalBoundsSpec := '[ TemporalBound#1 TemporalBound#2 '] ;
VariableTemporalBoundsSpec := LEXEME <| DDLPARAMETERVARIABLEFILTER ;
TemporalBound := TemporalBoundInteger | TemporalBoundSymbol ;
```

```
(AND
  (meets
   (SINGLE ((SEP_Schedule SEP_Schedule_SV))
   (Thrust_Segment (?_any_value_ ?_any_value_)))))
  (met_by (SINGLE ((SEP_Schedule SEP_Schedule_SV))
                  ((Thrust_Segment (?_any_value_ ?_any_value_)))))))
```

Figure 8 – top: fragment of grammar for DS1 constraint language; bottom: fragment of a constraint

A desirable objective is that planner experts, guided by the translations of their planner constraint language, would readily see how to use and write additional translations. Perhaps they could even go on to use the same approach to extend the planner constraint language itself, i.e., to automatically translate the formal expression of a requirement into the set of simpler constraints that the planner language currently accepts.

Applicability: when is translation a central problem, and how can it be accomplished?

Translation appears to us to be ubiquitous when seeking to perform analysis. The notations that people use for expressing requirements, designs, implementations, etc., are rarely the same notations that analysis tools accept as input. Translation bridges this gap. As standard notations come into more widespread use, people build translators from those notations to appropriate analysis tools. For example, (Mikk et al., 1999) describe a translator from Harel's Statechart notation into SPIN (input language of the model checker Promela). Prior to such standardization, however, people find the need to build domain-specific translators to go between domain-specific, even problem-specific, notations.

As indicated above, we believe construction of the translators themselves can take advantage of translation-building tools. A substantial example of this is in (Reyes & Richardson, 1998), where the authors employ (Reasoning SDK™) to prototype a domain-specific translator from test specifications to test drivers. Knowledge-based systems are particularly suited to these approaches, because of necessity they work with formal, machine-manipulable, notations. Their inputs and outputs are in a restricted notation, for which a formal grammar can readily be constructed.

9. Conclusions

Our work follows the trend towards the use of automation for generation of test automation. Specifically, our efforts led to the development of an automated generator of automatic test oracles. Motivations, issues and approaches to automatic test oracles and their construction are presented in (Richardson et al., 1992). The viability of their overall approach has been demonstrated by other studies, for example (Jagadeesan et al., 1997) presents an industrial application feasibility study on automatically constructing testing software for safety properties. Our work can be viewed as another confirming instance, one that has been carried through from conception to application. Additionally, our work brings to light some further areas of concern or emphasis, notably:

- The dominant concern was the limited resource of domain experts' time, *not* the efficiency of the test oracle itself. We

find that in much of the work published on test oracles, efficiency (and therefore scalability) of the test oracles themselves is a dominant concern. Commonly, safety properties (typically expressed in some form of temporal logic) are turned into finite state machines whose construction ensures their efficiency of execution, for example Dillon & Yu, 1994). For our particular application, the pilot studies revealed that efficiency of the test oracles would not be a driving concern, and that our database-based approach to analysis would suffice. More important to us was the investment of effort that would be required of our domain experts, whose time was in short supply. This led us to automate the generation of test oracles from a domain-specific representation. Thus the domain experts' effort it would take to construct that generator became out dominant concern. Approaches that could reduce this kind of effort include the parameterized tableaus (Dillon & Ramakrishna, 1996), or the algebraic-signature based mappings of (Reyes & Richardson, 1998).

- The need to yield needed test results with finer distinctions than simply "passed" or "failed." Information about "passed" cases was useful to for test coverage analysis, and for ascertaining that the test had been passed "for the right reasons". Information about "failed" cases was useful to locate the relevant portions of the plan contributing to those failures, and so speed the domain expert in debugging what was going wrong during planning.
- For knowledge-based systems that expose a trace of their inference process, it is very worthwhile to extend test oracles to crosscheck that inference information against the inference results. This is useful both to lead to increased assurance of the correct operation of the system under test, and to assist in the development of the test oracles themselves.
- We exploited the relative computational simplicity of checking vs. planning - an instance of Blum's notion of "simple checker" (Wasserman & Blum, 1997). As discussed above, we made do with computationally expensive test oracles, giving us the freedom to use a relatively general specification language from which oracles could be automatically derived. This gave us the grounds to *predict* that our approach to building the test oracles would suffice. We were able to *extend* the oracles (and the generation of those oracles) to accommodate additional checks with little additional cost. Finally, we were able to introduce an element of *validation* by offering a formal specification language of more generality than the domain-specific language itself; this permitted the more direct statement of intent, from which test oracles could be automatically generated.
- The complexity of test oracles for knowledge based systems stems from a different set of problems compared with test oracles for traditional software systems. Knowledge based systems, as exemplified by the DS-1 planner, yield results that can have complex structure. Much of the complexity of the test oracle is concerned with the extraction of information from that complex structure in order to judge its correctness. For example, the DS-1 planner yields plans; the conditions to be checked by the planchecker oracle concern the states the spacecraft would pass through were those plans to be executed. In contrast, oracles that operate on traditional software systems have access to the execution state of the systems, and/or to the output of that execution. Test methods founded upon the use of assertions embedded in the code, or on state machines driven by test logs, do not appear to be as useful for testing of planner-like systems. Conversely, knowledge based systems do not exhibit as wide a gap between the terminology of the specifications and the terminology of the executions.

We have based our lessons learned on our experience developing test automation for a spacecraft's autonomous planner. We see this specific application as a domain-specific instance of a wide class of knowledge based systems. Such systems pose both challenges and opportunities to V&V. The challenges arise because they are typically critical systems, and because the range of possible behaviors they may exhibit is very large. Thorough V&V is required but daunting. The opportunities arise because they adopt knowledge-based approaches, in which the data they manipulate is well structured and the purposes they fulfill are explicit. The opportunities make possible the vastly increased use of automation in V&V.

10. Acknowledgements

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

The authors thank the other members of the DS-1 planner team, especially Nicola Muscettola and Kanna Rajan, for their help.

11. References

J.F. Allen, 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832-843.

J.H. Andrews, 1998. Testing using Log File Analysis: Tools, Methods, and Issues. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering* (Honolulu, Hawaii, October 1998), IEEE Computer Society,

157-166.

D. Cohen, 1989. Compiling Complex Database Transition Triggers. *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Portland, Oregon, 1989), ACM Press, 225-234.

L.K. Dillon & Y.S. Ramakrishna, 1996. Generating Oracles from Your Favorite Temporal Logic Specifications. *Proceedings 4th ACM SIGSOFT Symposium Foundations of Software Engineering* (San Francisco, October 1996), ACM Press, 106-117.

L. Dillon & Q. Yu, 1994. Oracles for checking temporal properties of concurrent systems. *Proceedings 2nd ACM SIGSOFT Symposium Foundations of Software Engineering* (New Orleans, December 1994), ACM Press, 140-153.

B.P. Douglass 1998.. *Real-Time UML. Developing Efficient Objects for Embedded Systems.* Addison-Wessley.

DS1, 1998. http://nmp.jpl.nasa.gov/ds1/

M.S. Feather, 1998. Rapid Application of Lightweight Formal Methods for Consistency Analyses. *IEEE Transactions on Software Engineering*, 24(11): 949-959, Nov.

M.S. Feather & B. Smith, 1998. V&V of a Spacecraft's Autonomous Planner through Extended Automation. *Proceedings of the 23rd Annual Software Engineering Workshop* (NASA Goddard, MD, Dec. 1998).

W.J. Gutjahr 1999. Partition Testing vs. Random Testing: The Influence of Uncertainty. *IEEE Transactions on Software Engineering*, 25(5), Sep/Oct.

L.J. Jagadeesan, A. Proter, C, Puchol, J.C. Ramming & L.G.Votta, 1997.. Specification-based Testing of Reactive Software: Tools and Experiments. *Proceedings of the 19th International Conference on Software Engineering* (Boston, MA, May 1997), 525-535.

A .K. Jonsson, P. H. Morris, N. Muscettola, K. Rajan, & B. D. Smith *et a, l.*, 2000. Planning in Interplanetary Space: Theory and Practice. *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems* (Breckenridge, Colorado, April 2000), AAAI Press, 177-186.

Mars Climate Orbiter, 1999. *Mars Climate Orbiter Mishap Investigation Report*, Nov 10, 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf

NMP, 1999. http://nmp.jpl.nasa.gov/ds1/tech/autora.html

E. Mikk, Y. Lakhnech, M. Siegel & G. Holzmann, 1999. Implementing Statecharts in Promela/SPIN. *Proceedings of the 2nd IEEE Workshop on Industrial-Stength Formal Specification Techniques*, IEEE Computer Society, 1999, 90-101.

B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P.P. Nayak, M.D. Wagner & B.C. Williams, 1996. A Remote Agent Prototype for Spacecraft Autonomy. *Proceedings of the SPIE conference on Optical Science, Engineering and Instrumentation*.

B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P.P. Nayak, M.D. Wagner & B.C. Williams, 1997. An Autonomous Spacecraft Agent Prototype. *Proceedings First International Conference on Autonomous Agents*. ACM Press.

Reasoning SDK™. Reasoning, Inc. http://www.reasoning.com

A.A. Reyes & D.J. Richardson, 1998. Specification-Based Testing of Ada Units with Low Encapsulation. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering* (Honolulu, Hawaii, October 1998), IEEE Computer Society, 22-31.

D.J. Richardson, S.L. Aha & T.O. O'Malley, 1992. Specification-based Test Oracles for Reactive Systems. *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia, May 1992), 105-118.

B.D. Smith, M.S. Feather & N. Muscettola, 2000. Challenges and Methods in Testing the Remote Agent Planner. *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems* (Breckenridge, Colorado, April 2000), AAAI Press, 254-263.

H. Wasserman & M. Blum, 1997. Software Reliability via Run-Time Result-Checking. JACM 44(6): 826-845.

D. Wile, 1997. Abstract Syntax from Concrete Syntax. *Proceedings of the 19th International Conference on Software Engineering* (Boston, MA, May 1997), 472-480.